Strathprints Institutional Repository

Data Remanence and Digital Forensic Investigation for CUDA Graphics Processing Units

Xavier Bellekens*, Greig Paul*, James M. Irvine*, Christos Tachtatzis *, Robert C. Atkinson *
Tony Kirkham †, Craig Renfrew †
*Strathclyde University
{name.surname}@strath.ac.uk
†Keysight Technologies
{name_surname}@keysight.com

***Abstract*—This paper investigates the practicality of memory attacks on commercial Graphics Processing Units (GPUs). With recent advances in the performance and viability of using GPUs for various highly-parallelised data processing tasks, a number of security challenges are raised. Unscrupulous software running subsequently on the same GPU, either by the same user, or another user, in a multi-user system, may be able to gain access to the contents of the GPU memory. This contains data from previous program executions. In certain use-cases, where the GPU is used to offload intensive parallel processing such as pattern matching for an intrusion detection system, financial systems, or cryptographic algorithms, it may be possible for the GPU memory to contain privileged data, which would ordinarily be inaccessible to an unprivileged application running on the host computer. With GPUs potentially yielding access to confidential information, existing research in the field is built upon, to investigate the practicality of extracting data from global, shared and texture memory, and retrieving this data for further analysis. These techniques are also implemented on various GPUs using three different Nvidia CUDA versions. A novel methodology for digital forensic examination of GPU memory for remanent data is then proposed, along with some suggestions and considerations towards countermeasures and anti-forensic techniques.**

I. Introduction

Due to the exponential growth of data analytics, data storage, and network link speeds, Graphics Processing Units (GPUs) have been used by researchers as cost-effective off-the-shelf High Performance Computers (HPC). Various works have shown their efficacy in Intrusion Detection Systems (IDS) [1], Deep Packet Inspection (DPI), pattern matching [2], as well as database processing. These applications leverage the highly parallel processing capabilities offered by General Purpose Graphic Processing Units (GPGPU) but do not ensure the confidentiality of the data processed.

Inherently, GPUs are used to offload computationally intensive tasks from the CPU. Some of these tasks may require the processing or handling of confidential information, such as cryptographic keys [3], private network traffic [4] or financial data [5]. GPUs are also marketed as cloud computing services (GPU-as-a-service) where infrastructure and resources are shared between multiple tenants. Finally, personal GPUs and GPU-as-a-service can also be used by malicious actors for password cracking [6], network information retrieval, browser information retrieval [7] and to conceal malware [8][9]. All of these raise security issues pertaining to data remanence and digital forensic traces.

With such wide-ranging uses and applications of GPGPU technologies, there is naturally considerable interest in attempts to gain access to residual data on GPU memory. This may have been stored by a previous application running on the same GPU. In multi-tenant architectures, where one GPU may be used by multiple users on a time-share basis, this risk is raised since users will by definition be sharing the same GPU (and thus GPU memory) with other users, not all of whom may be respecting the privacy of the other users of the system [10].

Given the multitude of uses for GPU-based processing of sensitive or otherwise restricted data, this work aims to build upon the work by Breß [11], Di Pietro [12] and Maurice [10], and investigate the viability of accessing all types of GPU memory, on various different GPU platforms, including consumer, professional and mobile grade products. With more and more technologies taking advantage of GPU acceleration, including games and other proprietary data analytics solutions, the ease with which a rogue application may abuse access to previous memory contents is significant.

In this paper, data remanence and digital forensics methods, as would be applied to GPUs, are discussed. Data remanence is explored on Nvidia server-level, consumer-level and mobile-level GPUs. The contributions in this paper are three-fold:

- An assessment of different GPUs (and associated APIs) is provided, including different GPU architectures, Compute Unified Device Architecture (CUDA) architectures, and CUDA versions.
- Prior work [11], [12] has demonstrated the possibility of data retrieval from GPU global memory, however the authors found that shared memory was zeroed, making its contents inaccessible. This work utilises their approach, extending their experiments by demonstrating data retrieval for all memory types, including shared and texture. Additionally, this methodology is applied to multiple GPU types throughout the Nvidia range including high-end and mobile devices, indicating the high threat impact of data remanence.
- A digital forensics investigation methodology is provided, and anti-forensic countermeasures are investigated, for use on graphics processing units.

Section II of this paper describes the CUDA architecture, and Section III describes our experimental setups. Section IV describes and explains the results obtained, and the implications

for forensics. Section V is an overview of a novel forensic methodology for GPU memory, while Section VI describes potential countermeasures investigators could encounter. The conclusions and future work are outlined in Section VII.

II. GPU Environment

A. CUDA Programming Layer

CUDA is a parallel computing programming model developed by Nvidia. It allows the full computational power of GPUs to be harvested, and enables consumer grade devices to act as High-Performance Computers (HPCs). The massively parallel processing capabilities of Nvidia GPUs are present in off-the-shelf devices such as servers and consumer grade laptops, as well as mobile devices. CUDA also allows users to access their GPUs through a flexible abstraction model using the C/C++ programming languages [13].

To access the GPU, the C/C++ programming languages have been extended by Nvidia with a new set of instructions, libraries and directives, exposing the GPU hardware to users. The source code is split into host (CPU) and device (GPU) code. When compiled, the device code is translated into Parallel Thread Execution (PTX) code, which exposes the GPU as a processor, capable of carrying out the same operation on different data inputs simultaneously. The PTX code is then compiled into a *CUBIN*. The *CUBIN* code is a device-specific binary, optimised for the specific GPU architecture in use [14] [15].

The code running on the GPU is called a *kernel*, which is loaded by the host CPU. The execution of the kernel follows six steps: I) The host allocates memory on the GPU by using the set of instructions provided; II) The data is copied from the host memory to the GPU global memory; III) The host launches the kernel on the GPUs, if the system is composed of multiple GPUs, one can be specified; IV) The GPU executes the code in a single instruction multiple-data fashion; V) The computed results are transferred back from the device to the host; VI) Device pointers are de-allocated;

Threads launched by the *kernel* are organised in thread blocks, and each streaming multiprocessor (SM) can execute one or more thread blocks concurrently. For more granularity, threads within a thread block are organised in groups of 32 called a *warp*. The SM warp scheduler executes warps in a round robin fashion to maximise the resource utilisation on the GPU [16].

B. Hardware Memory Hierarchies

To maximise the performance of the GPU, several types of memory can be used by CUDA software to take advantage of the high performance of the GPU for data processing.

a) Global Memory: This is the main memory of the GPU, which can be both read and written to by the CPU and GPU. This memory is also known as on-board memory and is shared between all the stream processors, and therefore by all kernels running on the GPU. Global memory is also by far the largest memory on the GPU and can be accessed by 32, 64 or 128 byte memory transactions. All access to global memory should be coalesced, in order to allow *warps* to perform only a single memory transaction in a contiguous memory region, and therefore reduce the access latency [16].

b) Shared Memory: This is a 64 KB area of memory allocated per stream processor, shared with the L1 cache. Either 16 KB or 48 KB can be allocated as shared memory, with the remainder required for the L1 cache. The shared memory has an access bandwidth approaching 1.5TB/s, as this is on-chip memory. The shared memory is designed in a bank-switched fashion, with 16, 32 or 64 bit bank widths available, depending on the hardware. G200 GPUs use 16-bit banks, Fermi GPUs use 32-bit banks, and Kepler allows 64 bit banks (which permits storage of double precisions values) [17]. Shared memory is used by the users to efficiently share data across thread blocks, however when a *warp* accesses the same bank, a broadcast mechanism is triggered within the warp [15]. Shared memory is explicitly managed by the user and can be read and written by the GPU [14].

c) Local Memory: This is on-chip cached memory that can be read and written by all threads in a block. The memory is an abstraction of global memory and therefore has the same access time. Local memory is used to hold automatic variables, and is only used to spill data out of the registers. At *kernel* launch, if more memory is required, the CUDA driver will allocate memory on the fly, requiring extra time. Access to local memory must be fully coalesced allowing *warps* to access contiguous memory. Each access is cached through the L1 and L2 cache however only the L1 cache size can be increased or decreased based on the *kernel* requirements [14] [18].

d) Constant Memory: This is a read-only memory used for broadcasting data to all threads in a warp run by the GPU. The broadcasting take place in a single cycle. The memory is limited to 64 KB, however more can explicitly be requested by the user. Constant memory can only be written by the host. Data declared constant will reside in global memory, but is accessed via a specific set of instructions allowing an 8:1 ratio of cached data. As with global memory, Constant memory is off-chip unless the data accessed is cached [15].

e) Texture Memory: Texture memory is bound to global memory, however texture memory can only be accessed through specific hardware on the GPU. The memory is read-only and is optimised for 2D spatial locality. Texture memory also possesses a 8 KB cache per stream processor, allowing a considerable number of fetch operations to be saved. It can only contain fixed types such as *integer*, *float* and *char*, making texture memory the least flexible memory on the GPU. By default the textures are only accessible using floating point coordinates, and can only be accessed through a specific set of instructions, or an array bound to the texture memory [15] [16].

C. GPU Bidirectional Attacks

Graphics Processing Units are also subject to vulnerabilities that can be used to harm the end users of a system such as GPU-as-a-service.

Vasiliadis et al. demonstrate how to enhance the robustness of malware by taking advantage of the different GPU memory and by hiding the malware in global memory. The paper also describes a methodology to unpack polymorphic malware and techniques to recover and analyse data [8].

Ladakis et al. describe a key-logger software hosted on the GPU. The software takes advantage of the GPU by reading

the DMA channel, allowing the key-logger to analyse recorded keystrokes by storing them in GPU memory [9].

Breß et al. characterised a forensic methodology to retrieve data from GPUs used for database co-processing. The methodology depicts how a malicious user can bypass access controls and access data stored in the database. The study advises to clear the contents of memory after use [11].

Data leakage from GPUs in virtualised environments has been described by Maurice et al. in [10] where they detail how to bypass isolation mechanisms when using virtual machines. The paper reproduces the data leakage described in [11] and extends it with a detailed analysis of GPUs data leakage in the cloud.

Di Pietro et al. demonstrate the vulnerabilities of GPU architectures by reproducing the work of [11] and extending it to shared memory, and registers. They also demonstrate an attack against current cryptographic applications taking advantage of the power of the GPU [12].

These works demonstrate that GPUs can be used against the user, but also demonstrate the vulnerabilities related to specific parts of the graphics processing units used in the cloud or against one type of GPU. In our work, the experiments of [11] and [12] against global memory and shared memory are reproduced on consumer, mobile and professional grade GPUs. The work is then extended by using different versions of CUDA, by extending the technique to texture memory on the three devices, and by establishing a digital forensic methodology adapted to GPU that can be used in forensic investigations without requiring the long process of analysis, and reverse engineering of the drivers, as proposed in [10].

TABLE I. ARCHITECTURES USED FOR THE EXPERIMENTS

GPUs	GTX 295	Tesla K20m	Tegra K1
Number of GPUs	2	1	1
CUDA Version	5.5, 6.0, 6.5	5.5, 6.0, 6.5	6.0
CUDA Capabilities	1.3	3.5	3.2
GPU Architecture	GT200B	Kepler GK110	Kepler GK20a
Warp Size	32	32	32
CUDA Cores	8	192	192
Multiprocessors	30	13	1
Global Memory	896 MB	5 GB	2 GB

III. EXPERIMENTAL ENVIRONMENT

The experiments described in this paper were performed on three different Nvidia architectures as shown in Table I, covering the three main ranges of Nvidia GPU products, with default ECC settings. The GTX 295 is a consumer device containing two discrete GPUs - each GPU has 896 MB of global memory. The Tesla K20m is a high-end GPU used in HPC, with a single GPU and 5 GB of global memory. Finally the Jetson TK1 card contains Nvidia's next generation mobile GPU based on the Kepler architecture, which can be found in mobiles, laptops and cars. The TK1 GPU shares 2 GB of RAM with the Jetson development board's ARM-based CPU.

The experiments were carried out with network traces specifically generated for the purpose, which were stored on the GPU memory. The traces contained easily identifiable strings allowing the identification of data remanence in the different types of memory. This emulated the use of a pattern matching algorithm running on the GPU, storing confidential network traffic data during processing. Each experiment was repeated 100 times. In between each experiment, the memory under test was fully cleared.

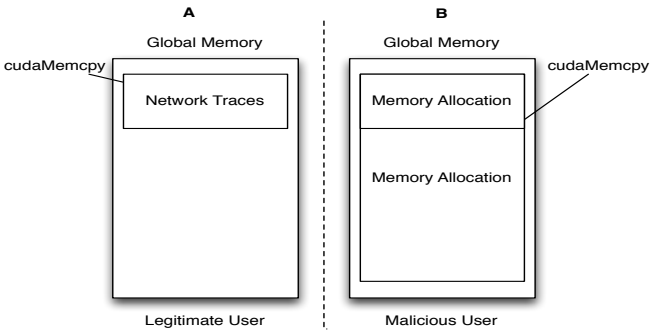

Fig. 1. Global Memory Data Remanence

IV. RESULTS AND EVALUATION

In this section, data remanence is evaluated on Global Memory, Shared Memory and Texture Memory, and analysed on the three different architectures described previously.

A. Global Memory

For testing Global memory data remanence, consider the scenario where the primary user utilises the GPU to analyse network traces through pattern matching algorithms. The user sends the network traces from the host to the GPU using *cudaMalloc* and *cudaMemcpy* as shown in Figure 1A. A second independent and malicious user then runs a secondary piece of software, requiring allocation of the same or a larger amount, of memory as that used by the primary user using *cudaMalloc* and dumps the remanent data stored from the GPU RAM to the host.

The memory allocation requested by the malicious user will allow partial or full recovery of the data stored within the GPU memory as shown in Figure 1B. The experimental results demonstrate that the choice of GPU, architecture or CUDA version does not affect the outcome, and that 100 percent of data remained in RAM, and was accessible for further use by other software running on the GPU. This vulnerability is due to the lack of secure software practices on the GPUs, as discussed in [10] [11].

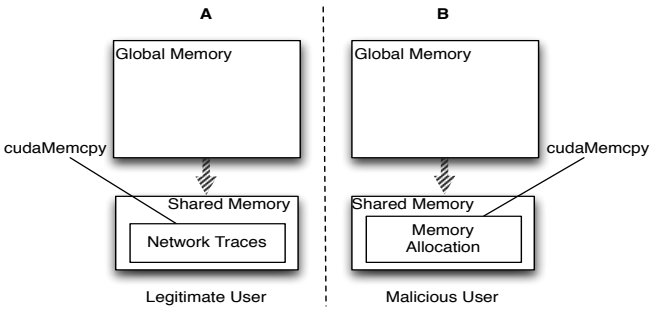

Fig. 2. Shared Memory Data Remanence

B. Shared Memory

In this experiment, synthetic network traces were sent to global memory, then stored in shared memory to be analysed as shown in Figure 2A. The legitimate user would run the network traces through the pattern matching algorithm until the results are sent back to the user-land. The size of the network traces corresponded to the size of the shared memory minus the size of the L1 cache (as memory is shared for both purposes).

A malicious user would then run a second process on the GPU requiring a maximum-capacity uninitialised array of shared memory. By using CUDA version 5.5, 6.0 or 6.5, the malicious user can use standard I/O to obtain the content of shared memory, or simply dump the content back to global memoryin Figure 2B. Di Pietro et al. state that shared memory is zeroed after execution [12], however in our experiments, the data remained on the GPUs after execution of the first user. These behaviours were observed on the consumer, server and mobile grade devices, and with the three different CUDA versions.

C. Texture Memory

For the texture memory experiment, the network traces are transferred from the CPU and bound into texture memory using *cudaBindTextureToArray* as shown in Figure 3A. The process for analysis of the data is the same as for the previous experiment.

The malicious user would then allocate the same size of array as defined by the legitimate user, or a significantly larger array, to be able to dump the memory contents, as shown in Figure 3B. When using CUDA 6.0 and 6.5, the data could be obtained via standard I/O 100 percent of the time, but could only be dumped to the host memory 20 percent of the time. When CUDA 5.5 was used, the data could be dumped to the host memory in 100 percent of the tests. This uncertain behaviour with CUDA versions 6.0 and 6.5 leads us to agree that this is not a security mechanism in place, rather a quirk due to the CUDA driver, as described in [12], [19].

D. Detailed Digital Forensics Testing

The extensive experiments carried out here demonstrate data remanence in the off-the-shelf graphic cards under various realistic situations, on different devices, using different kinds of memory. The results are organised into tables, *"Y"* meaning data remanence on the GPU was proven through recovery of the data, *"N"* means the attempt to recover data was unsuccessful and *"N/A"* stands for not applicable, where it was not possible to carry out a given test.

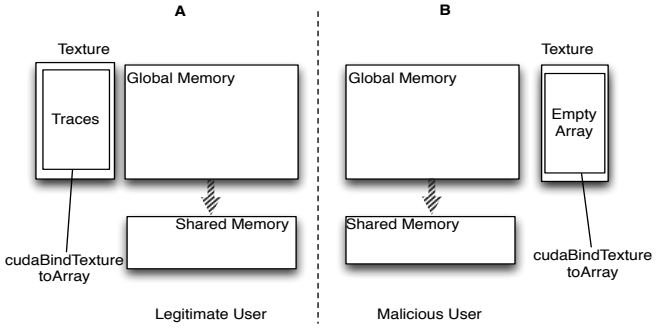

Fig. 3. Texture Memory Data Remanence

TABLE II. USER SWITCH

GPUs	GTX 295	Tesla K20m	Tegra K1
Global Memory	Y	Y	Y
Shared Memory	Y	Y	Y
Texture Memory	Y	Y	Y

Table II demonstrates and compares the behaviour of the three GPUs when a process is run by a malicious user after the legitimate user has run their software. These results also demonstrate the potential for digital forensics investigation on GPU when there are multiple users accounts on the computer and only one is accessible to the forensic investigator. The inherent sharing of memory at GPU-level, regardless of user-level separation on the host system allows access to remanent data.

TABLE III. GPU RESET

GPUs	GTX 295	Tesla K20m	Tegra K1
Global Memory	Y	Y	N/A
Shared Memory	Y	Y	N/A
Texture Memory	Y	Y	N/A

Table III shows the results of a GPU reset. This is carried out using the *nvidia_smi* tool available in the CUDA framework [10] [20]. This demonstrates that a reset of the GPU will not affect the memory contents on the consumer and server grade devices. The mobile device, however, does not have the ability to reset the GPU alone. These results demonstrate the possibility of data recovery in a forensic investigation when a suspect has attempted to wipe traces of previous activities through resetting the GPU using the built-in commands. These results also demonstrate that legitimate users should not rely on this function to avoid data leaks, or to prevent other users from accessing their data following processing on a shared GPU.

TABLE IV. HARD REBOOT OF THE HOST MACHINE

GPUs	GTX 295	Tesla K20m	Tegra K1
Global Memory	N	N	N
Shared Memory	N	N	N
Texture Memory	N	N	N

Table IV demonstrates that the GPU memory suffers from the same effect as Random Access Memory (RAM), where when a hard reboot is carried out, all data in RAM is lost. This occurs as data retention in the RAM requires power to be constantly applied to the device. As long as the GPU is powered, data can be recovered from the all three of the memory types on the GPU.

V. DIGITAL FORENSICS METHODOLOGY

Digital forensic investigators require a protocol and a methodology to follow, proceeding from the most volatile memory to the least volatile, allowing them to present significant evidence in court cases, in a reliable and verifiable fashion [21]; however, extracting shared memory occurs through global memory. In order to avoid overwriting any remanent content, global memory retrieval is prioritised.

Here we propose a novel methodology that shortens the investigation, allowing higher confidence in data integrity. The methodology presented in [10] requires investigators to reverse engineer the proprietary driver, which is a time consuming and a complex task, leading to uncertainty in the data integrity. In contrast, this novel methodology does not require reverse engineering of the CUDA driver, and follows the United States' Department of Justice recommendations that consists of "Preparation and Extraction" and "Identification" steps [22]. The novel methodology requires the controlled modification of program code running on the GPU while aiming to preserve the content of memory, this is not unlike mobile phone digital forensics where the requirements are similar [23]. In GPU and mobile scenarios, non-invasive approaches are not always possible due to technical limitations and requirements, and for this reason, there are a large number of variables in the process which need to be evaluated by an investigator.

Figure 4 depicts the appropriate steps a forensic investigator should follow before and during their investigation. As described in Section IV, the GPU must remain powered in order to be able to recover data. The first step should therefore always include physically securing access to the device in question, and ensuring the security of power supply to the device, as appropriate.

Preparation : This phase will require the forensic investigator to identify clearly the model of GPU, allowing them to understand the underlying architecture of the graphics card. This allows them to understand the types of memory in place, the amount of memory allocated to global, shared and texture memory respectively, and to uniquely identify the GPU in a multi-GPU, virtualised or mobile system.

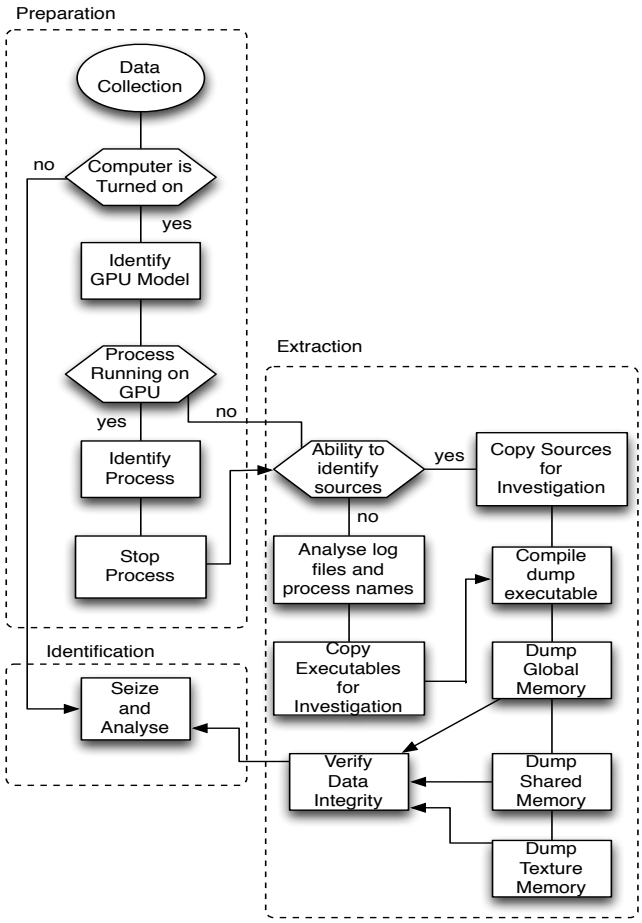

Fig. 4. Digital Forensics Methodology

Extraction : In the case where a process is running on the GPU, the process in question needs to be identified from the host operating system. Once clearly identified, the process can be stopped, such that its memory is free for further investigation. At this stage the investigator should identify the executable and the source code if available, and make copies of them for the analysis phase satisfying any other evidential requirements, such as retaining and comparing file checksums.

Based on the identification of the GPU and associated architecture in the "Preparation" phase, the investigator can now compile an appropriate *"dump"* executable, allowing them to access one or more GPUs on the system to recover Global, Shared and Texture memory. This compilation must take place on another computer, to prevent unintentional interference with potential forensic evidence on the computer under investigation.

Identification : The data retrieved from the running of the executable can be analysed by using standard forensic tools as well as freely available CUDA tools. In order to analyse the process sources for investigation, the full featured *Nsight* editor provided by Nvidia can be used, where simulations, profiling and in depth analysis can be performed [13]. Furthermore, the executables retrieved can be analysed and tested on sample GPUs to generate a behavioural analysis of the software. The executable can be further analysed through decompilation by using *nvdisasm*, *nvidia-debugdump* tools or *cuobjdump* [13]. Analysis of the data dumped from global, shared or texture can be analysed using standard memory analysis tools such as the sleuth-kit [24].

VI. Countermeasures and Anti-Forensics

As with any forensic analysis of volatile device memory, it should be noted that forensic analysis is constrained by whether or not the data in question remains to be retrieved, and that this is out-with the control of investigators in most circumstances. In this section, three potential technical countermeasures against forensics are considered.

A. Memory Overwriting

The most obvious means of mitigating threats revolving around recovery of data from GPU memory is to overwrite the memory contents, thus preventing techniques such as those discussed from recovering data successfully. By considering the performance implications of erasing GPU memory at time of initialisation, it is clear that for performance-optimised software, there remains motivation to avoid overwriting GPU memory, to reduce the performance overhead of erasing previous data from memory. These methods were proposed in [11] and [12].

B. Dynamic Parallelism

To extend this technique, dynamic parallelism can be used, requiring a *kernel* composed of one or more threads, which itself contains another *kernel*, responsible for the erasure of sensitive data following its access. This technique would reduce the required zeroing time, and can be easily implemented by programmers, or added in CUDA libraries, due to running within a second kernel separate from the original code.

Another potential technique is to plant misleading data within GPU memory, using a second kernel, running in parallel with the first, through dynamic parallelism. With two kernels writing, calculating, and modifying data in GPU memory, forensic examination would be made significantly more complex, due to the ease with which decoy data may be placed in memory, for the forensic examiner to deal with.

C. Dead Man's Handle Monitoring

The dead man's handle technique can also be used to run two processes on the GPU, which simultaneously monitor each other. In the event of one process being disturbed or terminated such as by someone attempting to terminate the process to begin dumping memory, the remaining process would initiate a memory-erasing routine, using the massively parallel hardware to wipe the entire GPU memory in a matter of seconds.

VII. Conclusions

In this work the data remanence on GPUs was examined for three different types of memory, three different CUDA versions and described for off-the-shelf server, consumer and mobile grade devices. It was demonstrated that users should consider data remanence when writing software on GPUs in un-trusted environments, such as shared servers and GPU-as-a-service. The experiments carried out demonstrated the retrievability of data within all three main types of GPU memory. A novel digital forensics methodology for GPU was presented, describing a guideline for forensic investigators to follow, shortening the investigation time while aiming at preserving the data stored on the GPU.

Finally, countermeasures and anti-forensics methods were described, providing a means for users to preserve the confidentiality of data they process on GPUs. Future works may include investigations of GPUs running on the Windows operating system, as well as investigation of the new Maxwell GPU architecture by Nvidia.

Acknowledgment

The authors would like to thank Keysight Technologies for their comments and feedback as well as their support.

References

[1] X. Bellekens, I. Andonovic, R. Atkinson, C. Renfrew, and T. Kirkham, "Investigation of GPU-based pattern matching," in *The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013)*, 2013.

[2] C.-H. Lin, C.-H. Liu, L.-S. Chien, and S.-C. Chang, "Accelerating pattern matching using a novel parallel algorithm on GPUs," *Computers, IEEE Transactions on*, vol. 62, pp. 1906–1916, Oct 2013.

[3] M. Alomari and K. Samsudin, "A framework for GPU-accelerated AES-XTS encryption in mobile devices," in *TENCON 2011 - 2011 IEEE Region 10 Conference*, pp. 144–148, Nov 2011.

[4] N.-F. Huang, H.-W. Hung, S.-H. Lai, Y.-M. Chu, and W.-Y. Tsai, "A GPU-based multiple-pattern matching algorithm for network intrusion detection systems," in *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on*, pp. 62–67, March 2008.

[5] M. Lee, J. hong Jeon, J. Kim, and J. Song, "Scalable and parallel implementation of a financial application on a GPU: With focus on out-of-core case," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pp. 1323–1327, June 2010.

[6] D. Apostal, K. Foerster, A. Chatterjee, and T. Desell, "Password recovery using MPI and CUDA," in *High Performance Computing (HiPC), 2012 19th International Conference on*, pp. 1–9, Dec 2012.

[7] S. Lee, Y. Kim, J. Kim, and J. Kim, "Stealing webpages rendered on your browser by exploiting gpu vulnerabilities," in *35th IEEE Symposium on Security & Privacy (S&P)*, 2014.

[8] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "GPU-assisted malware," in *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pp. 1–6, Oct 2010.

[9] E. Ladakis, G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "You can type, but you can't hide: A stealthy GPU-based keylogger," in *Proceedings of the Fourth European Workshop on System Security*, 2013.

[10] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "Confidentiality issues on a GPU in a virtualized environment," in *FC 2014, 18th International Conference on Financial Cryptography and Data Security, 3-7 March 2014, Barbados*, (Barbados, BARBADOS), 03 2014.

[11] S. Breß, S. Kiltz, and M. Schäler, "Forensics on GPU coprocessing in databases - research challenges, first experiments, and countermeasures," in *Datenbanksysteme für Business, Technologie und Web (BTW), - Workshopband, 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, pp. 115–129, 2013.

[12] R. Di Pietro, F. Lombardi, and A. Villani, "CUDA leaks: Information leakage in GPU architectures," *arXiv:1305.7383v2 [cs.CR]*, 2013.

[13] Nvidia, "Cuda C programming guide," 2013. http://docs.nvidia.com/cuda/.

[14] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[15] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU Computing Series, Morgan Kaufmann, 2013.

[16] R. Farber, *CUDA Application Design and Development*. Applications of GPU computing, Morgan Kaufmann, 2011.

[17] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education, 2010.

[18] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2012.

[19] F. Lombardi and R. Di Pietro, "Towards a gpu cloud: Benefits and security issues," in *Continued Rise of the Cloud* (Z. Mahmood, ed.), Computer Communications and Networks, pp. 3–22, Springer London, 2014.

[20] Nvidia, "Nvidia SMI," 2011. https://developer.nvidia.com/sites/NVML_cuda5/nvidia-smi.4.304.pdf.

[21] B. D and K. T, "Guidelines for evidence collection and archiving," 2002. https://www.ietf.org/rfc/rfc3227.txt.

[22] Department of Justice, "Digital forensics analysis methodology," Aug. 2007. http://www.justice.gov/criminal/cybercrime/docs /forensics_chart.pdf.

[23] N. Son, Y. Lee, D. Kim, J. I. James, S. Lee, and K. Lee, "A study of user data integrity during acquisition of android devices," *Digital Investigation*, vol. 10, Supplement, no. 0, pp. S3 – S11, 2013. The Proceedings of the Thirteenth Annual {DFRWS} Conference 13th Annual Digital Forensics Research Conference.

[24] K. Jones, R. Bejtlich, and C. Rose, *Real Digital Forensics: Computer Security and Incident Response*. Addison-Wesley, 2006.